Repeatable Reverse Engineering with PANDA

Brendan Dolan-Gavitt*, Josh Hodosh†, Patrick Hulin†, Tim Leek†, Ryan Whelan†

(Authors listed alphabetically)

*NYU
brendandg@nyu.edu
†MIT Lincoln Laboratory
{josh.hodosh,patrick.hulin,tleek,rwhelan}@ll.mit.edu

***Abstract*---We present PANDA, an open-source tool that has been purpose-built to support whole system reverse engineering. It is built upon the QEMU whole system emulator, and so analyses have access to all code executing in the guest and all data. PANDA adds the ability to record and replay executions, enabling iterative, deep, whole system analyses. Further, the replay log files are compact and shareable, allowing for repeatable experiments. A nine billion instruction boot of FreeBSD, e.g., is represented by only a few hundred MB. PANDA leverages QEMU's support of thirteen different CPU architectures to make analyses of those diverse instruction sets possible within the LLVM IR. In this way, PANDA can have a single dynamic taint analysis, for example, that precisely supports many CPUs. PANDA analyses are written in a simple plugin architecture which includes a mechanism to share functionality between plugins, increasing analysis code re-use and simplifying complex analysis development. We demonstrate PANDA's effectiveness via a number of use cases, including enabling an old but legitimately purchased game to run despite a lost CD key, in-depth diagnosis of an Internet Explorer crash, and uncovering the censorship activities and mechanisms of an IM client.**

I. Motivation

Reverse Engineering (RE) is the process of discovering undocumented internal principles of a piece of code. Why would anyone who is not a criminal want to do that? We can think of at least three socially acceptable uses for RE.

1) Enable legacy code to continue to function.
2) Identify critical vulnerabilities.
3) Understand the true purpose and actions of code.

It is common for legacy code to stop working as the software ecosystem surrounding it evolves. In that event, and when corporate support has also long terminated, RE is the most cost-effective avenue to continued use. Via RE, the inputs and outputs, the dependencies and requirements can be enumerated in detail, and appropriate shims fashioned to be able to run the old code in a more modern environment.

Accurately identifying vulnerabilities is usually impossible without detailed RE knowledge. That is, you might be able to observe a segmentation violation indicating an out-of-bounds read or write, but how do you determine if it is exploitable and therefore a critical vulnerability? The answer is that you need to determine *what* is illegally read or written and often that data is produced and consumed by closed source programs, libraries, drivers, and kernel. Thus, without either performing RE or making use of the RE efforts of others, it is difficult to discriminate between unimportant bugs and critical vulnerabilities.

Vetting code to determine if it does what it is purported to do and nothing else is an important and difficult task. This is obvious and uncontroversial when the code is believed to be malware. However, we believe that this is an increasingly fine distinction. Consider a program written by a legitimate, large, US company. Imagine that this code performs a host of malicious actions such as stealing personal information and modifying system settings. None of this behavior is indicated in the documentation or advertising literature, nor is it clearly essential for the primary purpose of the software. How is this code functionally distinct from malware? This is not simply a thought experiment--in 2005, Mark Russinovich discovered that Sony BMG audio CDs were installing a rootkit onto millions of computers [28]. The Sony rootkit recorded information about users' computers to send back to Sony and hid every file on the system with a certain prefix; worse, their uninstaller allowed any web page to download and execute arbitrary code [17].

If we have the time, inclination, or mission we will want to RE code to satisfy ourselves that what it does is acceptable to us. If we were philosophers, we might claim this as a basic right. Instead, we merely indicate that, in some situations, it is necessary and important to be able to divine the true function of closed programs. The only practical way to do this is through RE.

RE through Dynamic Analysis

Much reverse engineering is done statically. Disassemblers and decompilers translate binary code into a form more easily read. Humans painstakingly navigate these representations, adding extensive annotations to ultimately reassemble a picture of how code and data operate at various levels. This process can be valuable and productive, but it is not the PANDA model. While it occasionally makes use of offline static analyses, PANDA is fundamentally a dynamic tool and is

Distrubtion A; Public Relase. This work is sponsored by the Assistant Secretary of Defense for Research & Engineering under Air Force Contract #FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

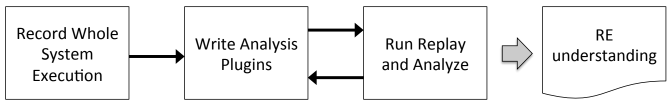


Fig. 1: Replay-based Reverse Engineering Workflow. PANDA's ability to record and replay whole system executions is the foundation of its use in reverse engineering. One captures a recording and then iteratively builds one or more plugins that perform dynamic analyses.

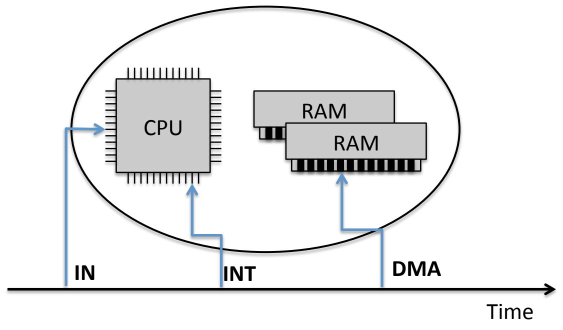


Fig. 2: PANDA Non-determinism log

used as depicted in Figure 1. First, one captures a recording of some whole system execution which one wishes to understand deeply. Then, one writes analysis code in the form of plugins, registering callback functions to be called by PANDA at opportune points such as before a basic block of guest code executes or at a virtual memory read. These plugins collect data and consult or control other plugins. Plugins are typically written quickly and iteratively, running the replay over and over to construct more and better and deeper understanding of the important aspects of system execution, given the RE task. An initial plugin might just get a rough outline of what processes execute and when key events happen. A second pass might focus in on the activity of a particular program or a portion of the replay. Further iterations might invoke a taint analysis that selectively labels interesting data and tracks it as it flows around the system. We have found that this workflow powerfully enables reverse engineering. The use cases section of this paper will detail three compelling examples of PANDA's use.

II. PANDA System

In this section, we describe the four main novel aspects of PANDA: its record/replay facility, its plugin architecture, its ability to use a single analysis for multiple architectures, and its ability to emulate Android systems.

A. Record / Replay

PANDA's record and replay facility is conceptually simple. The high-level view is depicted in Figure 2. We draw an imaginary line around the CPU and memory in the guest. At the beginning of recording, we take a snapshot of the machine state, which includes the contents of registers and memory.

Replay	Instructions	Log size	Instr/byte
freebsdboot.rr	9.3B	533MB	17
spotify.rr	12B	229MB	52
haikuurl.rr	8.6B	119MB	72
carberp1.rr	9.1B	43MB	212
win7iessl.rr	8.6B	9.4MB	915
game.rr	60M	1.8MB	33

TABLE I: ND log sizes for various replays

Then, we proceed to record to an *ND log* (non-determinism log) three kinds of inputs when they cross the imaginary line:

1) **IN**. The data entering the CPU on port input.
2) **INT**. A hardware interrupt and its parameters.
3) **DMA**. The data written to RAM during a direct memory access operation from a peripheral device.

When any of these inputs is written to the ND log, we also record the *trace point* which is enough information to determine when to replay the input. This trace point currently consists of three values: the program counter, the instruction count since record began, and the implicit loop variable (`ECX` on x86). These three are, in practice, sufficient to disambiguate trace points [16].

Note that this flavor of record/replay differs in one important respect from the one that used to be supported by, for instance, VMWare Workstation. There, the ND log consists of inputs to devices. Thus, VMWare's ND log can be used, during replay, to ``go live''. because the entire VM and all of its devices are always in a consistent state. PANDA does not execute any device code during replay, so there is no way to go live---but because our interest is in deep, replay-based analysis, we have no need to go live. In addition, our style has the virtue of simplicity, in that new architectures and devices can be supported with essentially no additional effort.

PANDA's replay is quite stable and effective. We have tested it on two architectures (32 and 64-bit x86 and ARM) extensively. It can record boot for a variety of operating systems, which is challenging given the complexity of that operation. It is also fairly compact despite the fact that our ND log must capture the contents of DMA inputs. Table I gives the ND log sizes for a number of architectures, workflows, and replay times. From this data, we can see that anywhere from about 20 to almost 1000 instructions execute per byte of log data. These replays cover a diverse set of tasks from booting FreeBSD to running the Carberp malware to installing a game. The modest size of these files makes them ideal for sharing, and is thus a major enabler of repeatable experiments. One of the authors has set up a web site where any of these and number of other replay files can be downloaded.[1]

Furthermore, the full repeatability of replays makes them incredibly useful for dynamic analysis. Traditionally, manual dynamic analysis involves running a program inside a debugger and using the debugger to introspect into program state. However, debuggers largely cannot execute backwards, so in order to inspect an earlier program state, an analyst must

[1]http://www.rrshare.org

Environment	Time in sec	Slowdown wrt Qemu 2.1.0
Qemu 2.1.0	35.6	1.0
PANDA	37.2	1.05
PANDA+record	66	1.85
PANDA+replay	127	3.57

TABLE II: PANDA, record, and replay slowdowns

restart the program from scratch. This will change every heap address used by the program, and without debugging symbols, every data structure location will have to be found manually. With PANDA replays, heap addresses are the same each time, so information about the state of memory can be built up piece by piece. This ability greatly accelerates reverse engineering.

Record and replay performance has not been a priority, as we have been focused on fidelity and system use. Consequently, it is currently fairly slow. We ran the configure and compile of the software `gzip-1.2.4` and the timing numbers are given in Table II. PANDA itself is about 5% slower than QEMU 2.1.0. Recording incurs a slowdown of almost 2x, and replay adds about another factor of 2. Thus, replay is almost 4x slower than standard QEMU. This may seem slow, but replay is noninteractive, and, in many cases, analysis plugins incur much larger overheads of 10-100x and so the replay slowdown is insignificant. The analysis benefits of replay are dramatic, as will be observed in Section IV.

B. Plugin Architecture

To understand PANDA's plugin system, it is first necessary to have a basic idea of how QEMU executes code in whole-system mode. As seen in Figure 3, code from the guest OS is initially translated by QEMU to an internal representation called TCG. If LLVM is not enabled, QEMU's JIT compiler will then generate host code and execute it, caching the generated code for future executions of the same basic block. When LLVM is enabled, the TCG IR is also translated to LLVM and the LLVM JIT can be used to generate and execute host code.

PANDA plugins take the form of shared libraries that can be loaded at any time during execution of guest code. When a plugin is loaded, its `init_plugin` function will be called to perform any setup; when the plugin is unloaded or PANDA exits, `uninit_plugin` will be called. In between, plugins are event-driven, performing work in response to things that happen in guest code.

Inside the initialization function, plugins can specify which events they want to instrument. The available instrumentation sites are located at many points throughout this process of guest code translation and execution. Plugins can be notified when an individual instruction is translated, and can signal that they would like to be notified whenever it executes ((1); labels in this paragraph refer to Figure 3). They can also be notified before and after translation of a basic block (2), and, if LLVM is enabled, analyze or transform the generated LLVM code (3). At runtime, plugins can receive callbacks before and after basic block execution ((4),(5)), and at memory access (6). There are a few other callbacks available to plugins, but they are more rarely used; the interested reader can consult the PANDA documentation for further details [2].

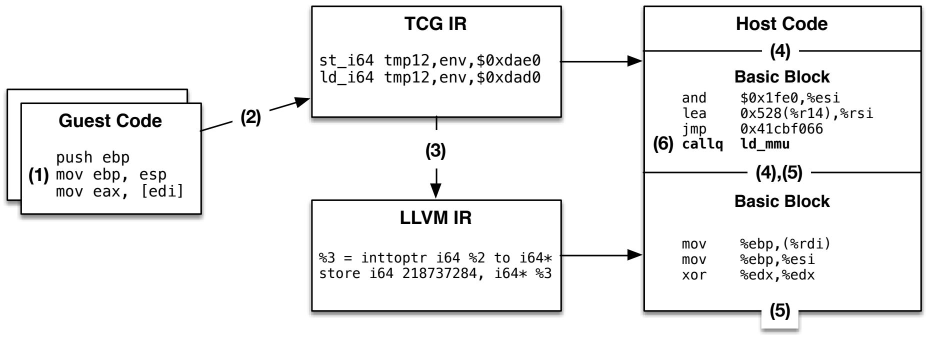
PANDA exposes an API that gives plugins access to some common functionality, including reading and writing memory, flushing the translated code cache, parsing command line arguments, and enabling various features such as memory instrumentation and precise program counter tracking that are too expensive to run all the time. In addition to this core API, plugins can call any public function in QEMU.

Many plugins depend on some common functionality. To avoid duplicating functionality throughout plugins while keeping the core of PANDA simple, we have implemented a mechanism for *plugin-plugin interaction*. This allows plugins to expose a public API that other plugins can call, and to define their own callbacks that other plugins can use to be notified of events. Although this functionality could be implemented by each plugin by using `dlopen` and `dlsym` to look up functions in other plugins, PANDA provides helpers that make exposing APIs and callbacks to other plugins easier and less error-prone. Through judicious use of preprocessor macro magic, plugins can define callbacks in just a few lines of code; other plugins can then sign up to be notified by invoking a single registration macro. Plugins can also make functions public by simply listing them in a specially named header file; PANDA's build system will then automatically generate an external header that allows other plugins to call them as though they were local functions. The precise details of the plugin-plugin interaction mechanism are available in the PANDA documentation [3].

Plugin-plugin interaction allows complex analyses to be rapidly built by composing existing plugins. For example, one can search for points in the execution that use a particular string using the `stringsearch` plugin (described in Section III-A), which defines a callback named `on_ssm` that triggers whenever a string match occurs. An analysis plugin might use this callback to be notified whenever a particular string (say, a password) is used. It could then taint the string in memory using the `taint` plugin (Section III-E) and register a callback with the `syscalls` plugin (Section III-B) that fires whenever `sys_send` is called in the guest and queries whether the data written is tainted. This would effectively implement a detector that could tell when data derived from a user's password is written to the network. Note the separation of concerns here: taint tracking, system call analysis, and string matching are all independent tasks relegated to their own plugins, while an analysis plugin coordinates their actions.

C. Architecture Neutral Analysis

There are a number of instruction-level dynamic analyses that are invaluable for reverse engineering. For instance, a taint analysis, in which data in memory or registers can be painted with labels and then those labels propagated through copies and collected in sets to represent computations, can permit detailed understanding of the true information flow patterns around and out of a system. But to perform this feat, dynamic taint requires an additional analysis be specified



```
push ebp
mov ebp, esp
mov eax, [edi]
```

```
st_i64 tmp12,env,$0xdae0
ld_i64 tmp12,env,$0xdad0
```

```
%3 = inttoptr i64 %2 to i64*
store i64 218737284, i64* %3
```

```
and    $0x1fe0,%esi
lea    0x528(%r14),%rsi
jmp    0x41cbf066
callq  ld_mmu
```

```
mov    %ebp,(%rdi)
mov    %ebp,%esi
xor    %edx,%edx
```

Fig. 3: Overview of PANDA execution and instrumentation. Detailed descriptions of the steps can be found in the main text.

and performed alongside every instruction to properly track labels. Another example would be a dynamic slice of the sort used by the Virtuoso [15] system to extract operating system introspection gadgets. This, likewise, necessitates per-instruction data-flow models. Third, consider the popular symbolic execution techniques, often coupled with SAT-solving. These are only possible once symbolic execution models have been implemented for every instruction. The QEMU emulator, upon which PANDA is built, supports over a dozen architectures. Completely specifying a taint analysis for just one of these is a daunting task, especially for x86 with over a thousand instruction variants.

PANDA avoids this pitfall by allowing every basic block of emulated code to be rendered in the LLVM intermediate language and thus for the analysis to take place in that simplified, but semantically equivalent domain. This means that, instead of having to write taint models for thousands of x86 instruction variants, we have to write models for only the few tens of necessary LLVM instructions. This is not only easier to get right, it also gives us equivalent analyses in all the architectures QEMU supports, with no additional work. LLVM translation is made possible via a module from the S2E system [8], which, in turn, leverages QEMU's own TCG intermediate language in which most guest instructions are implemented. We have modified and extended this LLVM translation to enable support for x86-64 and ARM architectures, and to ensure completeness through the inclusion of complicated instructions.

Many of the more complicated instructions such as MMX and SSE for x86 and floating point for ARM are actually relegated to C language ``helper'' functions in QEMU for performance reasons, and, presumably, because it would be tiresome to render them as TCG ops. These are potential analysis gaps which we bridge by generating LLVM for these functions via the `clang` compiler, and making those bitcode representations available for the same analyses as the basic blocks of emulated code. Thus, effectively all code whose business is the emulation of the guest can be submitted to something like a taint or symbolic execution analysis.

LLVM execution is currently slower than standard QEMU execution (this is also true for S2E, from which our LLVM execution is derived). We observed a slowdown of around 10x between a standard PANDA replay and a replay with LLVM enabled. We believe this could be improved by optimizing the LLVM code generated from TCG; previous work has even shown that it is possible to make QEMU *faster* by translating to LLVM [20].

D. Android Support

The Android SDK provided by Google includes a QEMU-based emulator, which emulates an ARM board named ``Goldfish''. This emulator is based on QEMU 0.8 (with some code backported from QEMU 0.10). We ported the features necessary to emulate Android devices to PANDA, jumpstarting our effort with the work Patrick Jackson did for the Google Summer of Code in 2011 attempting to integrate the Android code into mainline QEMU [21]. Significant additional work was required to be able to fully support modern Android emulation (through 4.4), including

- integrating telephony, camera, and ADB support
- integrating SD card support
- input translation for the VNC and SDL interface modes to Goldfish's input format
- supporting QEMU's QCOW2 disc image format for the storage devices
- arranging to support our record/replay mechanism
- fixing a video conversion bug

PANDA also incorporates code from the DroidScope project [35] for Linux/Android introspection. DroidScope is capable of tracking Linux processes, libraries, and threads, resolving debug symbols against a running Android system, and tracing the instructions run by a Dalvik interpreter in Android 2.3. We integrated the DroidScope code into PANDA and provided it with the necessary layout information for the Android 2.3 and 4.2 SDK kernels.

See Section IV-C for an example of the kind of deep reverse engineering of Android apps made possible by its integration into PANDA.

III. Plugin Details

Here we detail a few specific plugins commonly used when reverse engineering with PANDA.

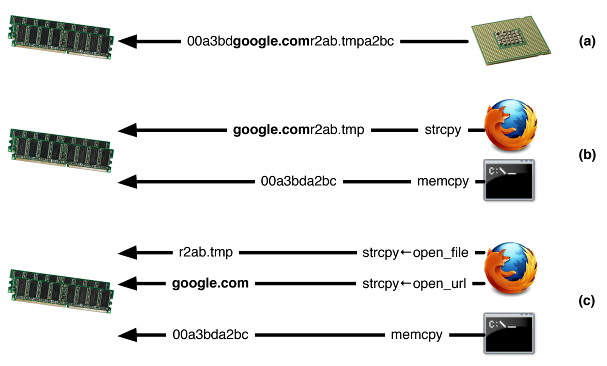


Fig. 4: Memory accesses made by a program with varying amounts of context: (a) as a single stream of information from the CPU to RAM ; (b) split up according to program and location within program ; (c) split up according to program, location within program, and calling context. Figure originally from [14].

A. Tappan Zee (North) Bridge

Reverse engineering tasks often hinge on finding out what piece of code implements some high level functionality or handles some particular data. In large programs this can be quite difficult: in the case where the data is some fixed string embedded in the program, it is easy enough, but for dynamic data one must usually laboriously trace the flow of data from some known input source through a chain of intermediate functions to where it is finally used. Moreover, in cases where the data sought is some intermediate value not directly derived from the input, even this approach may fail.

In previous work [14], we developed a system, Tappan Zee (North) Bridge (TZB for short), which attempted to locate points at which to interpose on a system for event monitoring during virtual machine introspection. We have since discovered that it is also immensely useful for reverse engineering.

The central idea of TZB is that memory accesses can illuminate the internal details of a system. As a program runs, functions called from different contexts read and write input, output, and intermediate results to memory. By appropriately separating out these memory accesses according to program and calling context (see Figure 4), we obtain coherent *streams* of data that can then be searched and analyzed for information of interest. We term these streams of data accessed at a particular point in a program *tap points* as they are places one might ``tap'' to get useful information from a system.

In the simplest case, we may wish to find out what part of a program handles a certain bit of data, such as a string we type into a program, or what function causes a particular string to be printed to the console or shown in the user interface. For such tasks we can just search all tap points for some fixed strings, which will give us a set of functions that were seen to read or write data matching our search string. To accomplish this we have created the `stringsearch` plugin, which tracks all memory accesses made in the system, splits them up according to the calling context, program counter, and address space, and then searches the resulting streams for a list of keywords.

In some cases we may not know the exact format of the data, but we may know some statistical features of it. For example, when searching for a DRM decryption function, previous work from Wang et al. [33] tells us that the inputs to such functions have high byte entropy and are statistically random (according to a test such as Pearson's Chi-Squared test), but their outputs are not random. We recently used this test to locate the DRM decryption function within Spotify and showed it could be used to extract unencrypted audio files [13].

To support this latter use case, PANDA can collect unigram and bigram statistics about each tap point using the appropriately named `unigrams` and `bigrams` plugins. These functions collect unigram and bigram histograms for data read or written at each tap point during an execution; once this summary data is gathered an analyst can write scripts to compute measures such as entropy, chi-squared values, or some distance measure (such as Kullback–Leibler divergence) to a previously observed distribution.

As we will see in Section IV, the ability to search through tap points for data of interest can allow a reverse engineer to quickly zero in on the parts of a program of greatest interest, or to extract normally unobservable data from a program as it runs. Note that TZB depends crucially on PANDA's record and replay functionality: trapping on every memory access and inspecting its contents on a live execution would cause it to grind to a halt, but the overhead is not a problem in a replayed execution.

B. System Calls

The `syscalls` plugin introspects on system calls in an emulated x86 or ARM guest. Upon encountering an instruction initiating a system call, it logs the instruction and system call number, and then dispatches on the system call number. This dispatch code is the heart of the plugin and is automatically generated from system call function prototypes. That is, it uses the known OS application binary interface along with the types in the prototype to make system call arguments available to callbacks for analysis. At each system call, the plugin logs the system call number and arguments, but then calls a function handler, passing system call arguments. By default, this handler is an empty function, but it can be overridden. For instance, we overrode several default handlers to build a file descriptor module which maintains a per-process mapping from open file descriptors to file names for Linux. The `syscalls` plugin also has an internal mechanism for executing callbacks when guest code returns from a system call. In our file descriptor tracker, this allows us to access data after it is read from a file to a buffer, detect the return code from system calls, copy the file descriptor list from a parent to child after a fork, etc.

Environment	Time (seconds)
PANDA+replay	3.6
PANDA+replay+LLVM	68
PANDA+replay+LLVM+taint	247

TABLE III: PANDA taint analysis slowdowns

C. Shadow Callstack

A common need in reverse engineering is to understand the context in which an event is happening. In particular, we often need to obtain the *callstack*---who called the current function, who called *that* function, and so on. In debugging contexts, this is usually done by walking the stack and reconstructing saved stack frames. However, this process is highly architecture and operating-system dependent, and programs can manipulate the callstack if they want to mislead analysis. This is even true of non-malicious programs like Windows, which tries to hide the callstack in its Kernel Patch Protection module [30].

To satisfy this need we include in PANDA a *shadow callstack plugin* named `callstack`. After each block in a replay executes, we check if it ended with a `call` instruction, and if so we push the return address onto the shadow stack. Before each block executes, we check to see whether it matches a return address on the callstack; if so, we know that the current function has returned and we can pop it from the stack. This approach requires the ability to disassemble blocks on a given architecture, but strong library support exists for that task.

The `callstack` plugin exports an API that allows other plugins to obtain the current stack of both return addresses and function entry points, and to register callbacks for function calls and returns. Together, these capabilities allow other plugins to quickly and accurately obtain information about the shadow callstack; furthermore, because the shadow stack is outside the control of in-guest programs, it is not susceptible to manipulation.

D. Scissors

One of PANDA's biggest contributions is the ability to do offline analysis: to collect a recording of execution at normal speed and then replay that execution with heavyweight analyses running, potentially over a long period of time. Still, though, many analyses are too heavyweight to be tractable over replays with potentially billions of instructions; examples include symbolic execution and complex taint analyses. To address this issue, we created the scissors plugin, which enables the user to excise smaller portions of replays and then analyze just the shortened portion. Combined with the ability of components such as TZB to rapidly locate sections of interest in the replay, the scissors plugin allows analysts to focus heavy attention on key events during execution and ignore everything else.

E. Taint Analysis

We have implemented a dynamic taint analysis as a plugin for PANDA. It permits precise labeling of data in a number of ways, including file contents, network input, RAM, and registers. These labels are then tracked automatically, and stored in a *shadow memory* that associates tainted physical addresses, registers, and I/O buffers with label sets. The propagation of labels is handled by inline LLVM code which executes a single taint operation corresponding to each LLVM instruction. These taint operations are derived from the LLVM translation described in Section II-C, via an LLVM ``pass'' which generates taint operations from intermediate language operations one by one. This means that our taint analysis is architecture-independent; we have used it to analyze x86, x86-64, and ARM replays, and we can trivially extend it to all of the architectures that QEMU supports. Finally, there are query mechanisms to determine if data is tainted at some replay point, and, further, to examine the set of associated taint labels.

Many of the details of how this subsystem was designed and implemented have been described elsewhere [34]. We have since reimplemented our taint processing with a focus on performance, and as a result, performance overhead is greatly reduced.

1) **Whole-system** PANDA's taint tracks labels even if they flow between processes including the kernel, and is indifferent to shared memory since the RAM portion of the shadow memory is in terms of physical addresses.
2) **Replay-based** Taint is an expensive analysis and for many platforms such as Android, even pure QEMU-based execution is barely fast enough to prevent time-outs. Therefore, the taint plugin can only run during replay.
3) **Detail and fidelity** The taint analysis in PANDA is focused more on the detail that can be obtained from the analysis rather than fast performance. For instance, a file can be labeled such that every byte in the file gets a different label. Further, guest computation is modeled with high fidelity via set unions. So, if `EAX` has the label set $1, 2$ associated with it in the shadow memory, and `EDX` has 3, then `EDX = EAX + EDX` will have the label set $1, 2, 3$.
4) **Performance** Such detailed analysis is inherently heavy-weight. Table III shows the slowdown of our system for one representative workload: tainting 13 bytes of data that are used in a product key verification check. Adding LLVM execution on top of QEMU's normal TCG intermediate representation typically adds 10-20x overhead, and adding taint analysis on top of LLVM adds another 3-10x overhead. The specifics are highly workload-dependent.
5) **Interface** Taint labeling and querying can be either event-driven (via callbacks registered with the taint plugin) or invoked by calling the plugin API. Users can easily use the file_taint plugin for system call-based tainting of file reads, and the tainted_branch plugin

allows identification of branches that use tainted data.

6) **IDA plugin** We have developed a plugin for the popular disassembler IDA Pro that allows visualization of instructions and functions which process tainted data.

IV. Case Studies

In this section, we present three compelling RE use cases for PANDA. In the first, an old PC game for which the CD key has been lost is rapidly made whole again by locating the key verification code and harnessing it to produce keys on demand. In the second, a Windows Internet Explorer vulnerability is diagnosed in depth from a whole-system replay, indicating not merely that it is a use-after-free bug but pointing the finger at a precise CVE number. In the third, an IM client suspected of censoring messages is quickly determined to be doing so via a blacklist which is also readily extracted. Note that, while we end up using many of the plugins mentioned in Section III, no attempt was made to cover all of them with our use cases. Rather, we allowed the task at hand to drive the plugins employed.

A. Reviving Legacy Code

We used PANDA to find and rapidly reverse engineer the 26-character CD-key validation algorithm for a popular video game from 1999. First, we collected a recording of the installer rejecting a random sequence of letters and numbers. We then provided both this incorrect key sequence and the text of the rejection dialog as searches to PANDA's TZB, which promptly found both in the replay. This focused our attention on about 200,000 instructions out of 60M in the complete replay (a reduction of 300x), and we used the `scissors` plugin to extract just this operative segment containing the validation algorithm.

Via manual static analysis of the code in the remaining replay segment, we ascertained that the installer decrypts the CD-key and checks the high-order bits of the resulting 120-bit integer against a fixed value. This magic number is not immediately apparent in the disassembly, but a trivial plugin was rapidly fashioned that printed it out when read from memory while running on the scissored replay. The magic number turns out to be 23. Manual reverse engineering from there easily revealed the complete key computation algorithm. Some additional mathematical analysis indicated a very low key density: only 1 in 27,000 of the possible keys are actually valid.

We then proceeded to extract source code for the key computation function, as the low key density indicated that harnessing it and trying random keys would be successful. QEMU's physical memory dump feature run at the end of replay plus the Volatility Framework [5] easily extracted the installer binary. IDA Pro's HexRays decompiler was then used to recreate source code which required only light editing and some error correction to compile. This extracted function we harnessed as an oracle in a small C program, feeding it random keys to find valid ones, which we then verified successfully unlocked the installer. It should be noted that, even without

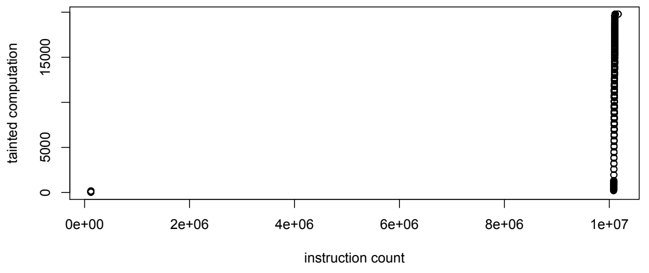


Fig. 5: Measure of tainted computation as a function of instruction count reveals where CD key validation computation occurs.

any manual reverse engineering, it would have been feasible in this case to generate a random key in this way. However, that might not have been the case. If the CD-key system had been designed for a lower key density, we would have had to invert the decryption algorithm. In this game's case, that would not be terribly difficult, but for other games it might be impossible. Binary patching or similar techniques would then have been necessary in order to play the game, which would be easy given the RE knowledge already assembled in this effort.

This RE effort was very successful. PANDA allowed us to extremely rapidly locate the code of interest, as well as find the comparison value of the test. The `scissors` plugin enabled us to reduce the replay to a size where complicated analysis was immediately tractable.

Additionally, we tried using a more complicated taint-based series of plugins for this RE task with very good results. We used TZB to apply taint labels to the CD key and then compute a measure of how much computation has taken place with tainted data. This measure is plotted in Figure 5 as a function of the instruction count in a portion of the replay for the game installer, and it clearly indicates a very small replay region of about 20,000 instructions where the key is decrypted and some bit-spreading takes place. Further, if we ask the taint system to identify the code responsible for those computations, the result is just twenty basic blocks of code. Of these, the block that performs the most computation is the one that decrypts the key. This taint-based analysis is powerful but it is not fast. The 4-second scissored replay that contains everything from first TZB match of CD key to seeing the invalid key dialog takes over 400 seconds to analyze and produce the graph in this paper.

B. Deep Vulnerability Diagnosis

Vulnerabilities often have deep causes, where the underlying bug can occur well before a potential crash or exploit. One classic example is the use-after-free bug in which a program retains a dangling pointer referencing freed memory, dereferencing it much later. Frequently this dereference will cause the program to crash by corrupting program or heap data structures, but the crash itself will give no hint about the dangling pointer---where it was created, when the memory was freed, or even that the bug involved a use-after-free at all.

As an exercise, we had one team member prepare a replay containing a triggered vulnerability. This replay was then given out with no information other than that an application crashed with the standard Windows error message, ``Application has stopped working.'' First, we used the `replaymovie` plugin to make a series of captures from the video card framebuffer and stitch them together into a video of replay execution. This indicated that the failing process was Internet Explorer, and that the vulnerability was triggered by loading an HTML file. We then used TZB to search for ``<HTML'' and ``has stopped working''; this gave us temporal bounds in the replay for where the bug must be. The `scissors` plugin enabled us to reduce the size of the replay and make heavier analyses. Using TZB again, we extracted all further output at the <HTML tap point, which was exactly the full HTML which triggered the bug. The HTML indicated that the vulnerability was probably a use-after-free.

We then wrote a custom use-after-free detector plugin for PANDA to locate the use-after-free. The detector was written for Windows, but could easily be adapted for other operating systems. It tracks calls to Windows's low-level RtlAllocateHeap, RtlFreeHeap, and RtlReAllocateHeap, and it maintains shadow lists of allocated and freed memory. When a pointer to freed memory is dereferenced, a use-after-free has occurred and the plugin detects it. This approach produces some false negatives (since a new allocation may have since occupied the free space), but in this case it successfully detected the use-after-free bug. Looking up the relevant function using Windows debug symbols quickly identified the bug as CVE-2011-1255.

In a real situation, the analyst would usually be given an HTML file which triggered the bug, but we felt this example would illustrate how PANDA's built-in plugins integrate well with custom analyses to give deep insight into system behavior. One person wrote the use-after-free detector in about 15 hours of working time and about 300 lines of code as measured by sloc. Further work could easily add full tracking infrastructure for dangling pointers, from creation to use. This addition would enable the detection of exploits as well as crashes.

In this case study, the repeatability of PANDA replays was a key advantage. It is much easier to write custom plugins when they are written to target a specific replay, as the author does not have to worry about writing the plugin in full generality; instead, they can hardcode addresses and trace points to gather information at key events.

C. Uncovering Censorship Blacklists

As we discussed in Section I, we cannot always trust that the software we use is acting in our interests. For example, many instant messaging clients actively censor the chats of their users [29], [25]. Such censorship can either be done at the server or by using a client-side blocklist that is periodically updated. In the former case PANDA can be of no help, since there is no code available to run and examine *in vivo*; however, in the latter we can use PANDA to extract a list of censored words from the client. We examined LINE messenger, which has been found [18] to censor users in some parts of the world. The CitizenLab analysis was done on version 3.8.5; we used the most recent version available, 4.5.4, which we downloaded from Google Play.

After installing LINE inside PANDA, we found that the registration process requires SMS verification; because we did not have access to a mobile number in the appropriate region, we signed up using a US phone number, which causes LINE to bypass its censored word checks. Referring to CitizenLab's previously published analysis, we were able to modify LINE's `sqlite` settings database and change our region to `CN`.

Once our region was correctly set, we created a recording in which we launched the LINE messenger and sent an instant message to another user. The message sent did not include any content we thought might be censored. Simply by sending the IM, we supposed that LINE would still have to load its censored words list and check our message against it, which would leave it open to extraction by PANDA.

To find the encrypted wordlist, we employed TZB, described in Section III-A. Guessing that a censorship list for China might contain terms such as ``Falun'' (法轮) and ``Tiananmen'' (天安门), we searched all memory reads and writes made by LINE for the UTF-8 encoded versions of these words. This gave us a set of tap points that contained the sensitive words. As we suspected, the words we sought were indeed included in LINE's list of censored words: four separate tap points contained both Falun and Tiananmen, along with 534 other words.

Because applications on Android are written in Java and JITed using the Dalvik virtual machine, the location of the code that handled the censorship blacklist was not particularly illuminating; in addition, the data passing through the tap point included a large amount of irrelevant data. To extract just the censored word list, we instead looked at the address of the memory that was accessed, and wrote a new plugin called `bufmon` that records every read and write to a given memory region. We then ran the replay a second time, providing `bufmon` the memory range we saw the censorship list being written to, and from its output obtained the full list without any extraneous information.[2] To save space, the censorship blacklist itself is omitted from this paper, but can be found at:

http://cc.gatech.edu/~brendan/line.txt

Had the words we thought of not been present in the censorship blacklist, we could have proceeded with a more sophisticated analysis. PANDA's record and replay functionality and plugin system allow us to run analyses of increasing sophistication over the same execution, zeroing in on the

[2]We also verified that the extracted list was correct by manually analyzing a decompiled version of LINE.

desired functionality or data present in the program we are reverse engineering. Such iterative analyses are tremendously simplified in PANDA compared to traditional, non-replay based dynamic analysis; for example, both the censorship wordlist buffer and the JITed Dalvik code stayed at the same memory address throughout the entire analysis, making it trivial to apply analyses to the same objects each time.

We also sidestepped some challenges by focusing on a dynamic analysis: LINE's censorship lists are encrypted with AES, and a static approach would have had to find the key embedded in the program in order to get the cleartext list. By looking at memory accesses, we were effectively able to ignore the encryption altogether. In all, once the region was correctly set in LINE, the entire analysis took about four hours, including the time to write and debug the `bufmon` plugin.

Reproducing these Results

We have designed PANDA so that analyses can be shared with others. Accordingly, we have made the replay logs available on rrshare.org, included the analysis plugins in our GitHub repository [1], and posted instructions for running PANDA to reproduce the use cases in this section at

http://cc.gatech.edu/~brendan/panda_re_repro.html

V. Related Work

We believe that PANDA's combination of repeatable dynamic analysis, powerful architecture-neutral analyses, and modular architecture form a uniquely powerful environment for reverse engineering. However, the individual components of PANDA are built on a rich body of work on dynamic binary instrumentation, record and replay, and taint analysis, which we survey in this section.

A number of systems have approached the problem of instrumenting binary programs at runtime. At the level of an individual program, systems such as Pin [26] and DynamoRIO [7] use dynamic binary translation to provide an API similar to PANDA's that allows users to instrument basic block execution, memory accesses, and so on. At the whole-system level, the BitBlaze project [31] (and in particular its TEMU component), S2E [8], and DECAF [19] are all QEMU-based and have functionality that overlaps with pieces of PANDA (indeed, some parts of PANDA are directly derived from these projects: our LLVM translation comes from S2E, and some of the Android introspection code is derived from DroidScope, the progenitor of DECAF). PANDA's novelty and effectiveness come from the focus on providing repeatable, modular analyses that can draw on sophisticated techniques such as taint without perturbing execution.

Record and replay itself is a well-studied research area in software engineering. The idea that replay can allow analyses to be decoupled from execution was first recognized by Chow et al. in Aftersight [9]; this insight is crucial to PANDA's model for reverse engineering. There have been a number of whole-system record and replay efforts prior to PANDA; the most prominent of these are VMWare's (now discontinued) record and replay system [32], TTVM [23], and ReVirt [16]. To the best of our knowledge, however, PANDA is the first open-source, widely available system that supports record and replay of multiple CPU architectures and provides compact recording logs that can be shared and reproduced on other machines.

The basic principle of tainting data and then following its propagation through a system has been explored at least since Perl's introduction of the ``-T'' flag to enable tainted variable checking in 1989 [4]. Under a somewhat broader interpretation of taint analysis, one could argue that earlier information flow models of security such as Bell-LaPadula [6] foreshadowed more recent analyses. The modern strain of research in taint analysis, however, begins in 2004, when several papers independently proposed tracking the flow of input data to detect exploitation of software vulnerabilities [10], [12], [27]. Subsequent research has focused on improving the efficiency and precision of taint analysis [22], [11]. Most recently, several of the authors of this paper designed the architecture-neutral taint analysis system used in PANDA [34]. Independently, Henderson et al. have also proposed a QEMU-based architecture-neutral taint analysis system [19]; PANDA's differs mainly in being based on LLVM rather than QEMU's own TCG and in its ability to follow taint through QEMU's ``helper functions'', which are implemented in C. This difference vastly enlarges the space of programs PANDA can analyze, as it can process floating point instructions and other processor extensions.

VI. Limitations and Future Work

PANDA currently has a number of limitations that we hope to address in future work. This section gives an overview of its main deficiencies and our plans to address them.

Performance: PANDA's responsiveness when interacting with a guest virtual machine during record has so far been adequate for our purposes, and its replay-based analyses do not require interaction. Nevertheless, we do hope to reduce record-time overhead. The primary source of overhead is the need to keep track of the number of instructions executed and the current program counter at the instruction level. It would be a moderate engineering effort to address this. A much more significant speedup could be achieved by recording under hardware virtualization via KVM [24] and only using CPU emulation during replay. This would require a significant implementation effort, however, as KVM does not natively support the mechanisms of our record / replay. Other inefficiencies might be more important to address, such as the 10x slowdown for execution under LLVM translation and the 25-100x penalty for taint analysis.

Architecture Support: Although PANDA is in principle architecture-neutral, there are a number of features that have not yet been ported to all architectures. The most significant of these is record / replay: although in principle supporting more architectures should be a simple matter of determining the architectural sources of nondeterminism and wrapping them in our existing record and replay mechanisms, we have not yet made this effort on all architectures. The highest priority cases are MIPS and PowerPC, as these CPUs are used extensively in embedded devices that we wish to analyze. LLVM translation is also not currently enabled for all architectures, but this will require much less work, as there are a relatively small number of changes needed to support LLVM translation and execution on other architectures.

Introspection: Some analyses may require additional, higher-level information about the state of the running system. For example, an analysis plugin might wish to restrict its attention to just one program or one module within a program; this requires some level of OS knowledge. We plan to create new plugins that encapsulate the domain-specific knowledge necessary to retrieve useful information about various guest OSes and expose that state via a public API, either by leveraging existing introspection tools such as Volatility [5] or relying on small custom-built introspection code of our own.

Truth be told, it is most likely that we will produce more analysis plugins to support interesting and complex reverse engineering scenarios before we address the above limitations. While it would be nice to have a slightly faster and more featureful PANDA, these limitations do not currently represent ``pain points'' that hamper our daily work, and we are far more interested in new applications of PANDA. Of course, as an open source project we welcome any contributions from the community.

VII. Conclusion

We have been actively using PANDA for the past two years to quickly reverse engineer large, real-world binary systems. In that time, we have found it to be invaluable for speeding up reverse engineering, in most cases either entirely obviating the need for manual analysis, or precisely directing human attention to the critical portions of a large code base.

The academic study of reverse engineering has been stalled by the widespread view that reverse engineering is a less than legitimate field of inquiry. It is our hope that with this paper we can begin the rehabilitation process. We have presented several compelling *prosocial* use cases for reverse engineering and provided a powerful new tool to facilitate good work. In doing so, we hope to encourage further research into tools and techniques that automate RE or make manual RE more effective. More fundamentally, though, we want to demonstrate that RE is a powerful tool for investigating the critical but opaque programs that increasingly run our world.

References

[1] PANDA: Platform for architecture-neutral dynamic analysis. https://github.com/moyix/panda/.

[2] PANDA plugin documentation. https://github.com/moyix/panda/blob/master/docs/PANDA.md.

[3] PANDA plugin-plugin interaction. https://github.com/moyix/panda/blob/master/docs/ppp.md.

[4] perlsec: Taint mode. http://perldoc.perl.org/perlsec.html#Taint-mode.

[5] Volatility: An advanced memory forensics framework. https://github.com/volatilityfoundation/volatility.

[6] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, MITRE Corp., Bedford, MA, 1973.

[7] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Cambridge, MA, USA, 2004. AAI0807735.

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 265--278, New York, NY, USA, 2011. ACM.

[9] Jim Chow, Tal Garfinkel, and Peter M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ATC'08, pages 1--14, Berkeley, CA, USA, 2008. USENIX Association.

[10] Jim Chow, Ben Pfaff, Tal Garfinkel, Kevin Christopher, and Mendel Rosenblum. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 22--22, Berkeley, CA, USA, 2004. USENIX Association.

[11] James Clause, Wanchun Li, and Alessandro Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196--206, New York, NY, USA, 2007. ACM.

[12] J. R. Crandall and F. T. Chong. Architectural support for software security through control data integrity. In *International Symposium on Microarchitecture*, 2004.

[13] Brendan Dolan-Gavitt. Breaking Spotify DRM with PANDA. http://moyix.blogspot.com/2014/07/breaking-spotify-drm-with-panda.html, July 2014.

[14] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan Zee (North) Bridge: Mining memory accesses for introspection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013.

[15] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, May 2011.

[16] George W Dunlap, Samuel T King, Sukru Cinar, Murtaza A Basrai, and Peter M Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. ACM, 2002.

[17] Ed Felten. Sony's web-based uninstaller opens a big security hole; Sony to recall discs. https://freedom-to-tinker.com/blog/felten/sonys-web-based-uninstaller-opens-big-security-hole-sony-recall-discs/, November 2005.

[18] Seth Hardy. Asia Chats: Investigating regionally-based keyword censorship in LINE. https://citizenlab.org/2013/11/asia-chats-investigating-regionally-based-keyword-censorship-line/, Nov 2013.

[19] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 248--258, New York, NY, USA, 2014. ACM.

[20] Ding-Yong Hong, Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, and Yeh-Ching Chung. HQEMU: A multi-threaded and retargetable dynamic binary translator on multicores. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 104--113, New York, NY, USA, 2012. ACM.

[21] Patrick Jackson. Upstreaming the Android Emulator. http://gsoc11-qemu-android.blogspot.com.

[22] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Network and Distributed Systems Symposium (NDSS)*, 2011.

[23] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proceedings of the 2005 Usenix Annual Technical Conference*, 2005.

[24] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225--230, 2007.

[25] Jeffrey Knockel, Jedidiah R. Crandall, and Jared Saia. Three Researchers, Five Conjectures: An Empirical Analysis of TOM-Skype

Censorship and Surveillance. In *Free and Open Communications on the Internet*, San Francisco, CA, USA, 2011. USENIX.

[26] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190--200, New York, NY, USA, 2005. ACM.

[27] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Network and Distributed Systems Symposium (NDSS)*, 2005.

[28] Mark Russinovich. Sony, rootkits and digital rights management gone too far. https://web.archive.org/web/20051102053346/http://www.sysinternals.com/blog/2005/10/sony-rootkits-and-digital-rights.html, October 2005.

[29] Adam Senft, Aim Sinpeng, Andrew Hilts, Byron Sonne, Greg Wiseman, Irene Poetranto, Jakub Dalek, Jason Q. Ng, Katie Kleemola, Masashi Crete-Nishihata, and Seth Hardy. Asia Chats: Analyzing information controls and privacy in Asian messaging applications. https://citizenlab.org/2013/11/asia-chats-analyzing-information-controls-privacy-asian-messaging-applications/, Nov 2013.

[30] Skywing. PatchGuard reloaded: A brief analysis of PatchGuard version 3. http://uninformed.org/index.cgi?v=8&a=5, September 2007.

[31] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Information systems security*. 2008.

[32] VMWare. Enhanced execution record/replay in workstation 6.5, 2008. http://blogs.vmware.com/workstation/2008/04/enhanced-execut.html.

[33] Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Steal this movie: Automatically bypassing drm protection in streaming media services. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 687--702, Washington, D.C., 2013. USENIX.

[34] Ryan Whelan, Tim Leek, and David Kaeli. Architecture-independent dynamic information flow tracking. In *Proceedings of the 22Nd International Conference on Compiler Construction*, CC'13, pages 144--163, Berlin, Heidelberg, 2013. Springer-Verlag.

[35] Lok-Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 29--29, Berkeley, CA, USA, 2012. USENIX Association.